RTker

Anubha Verma
Shikha Kapoor

Features

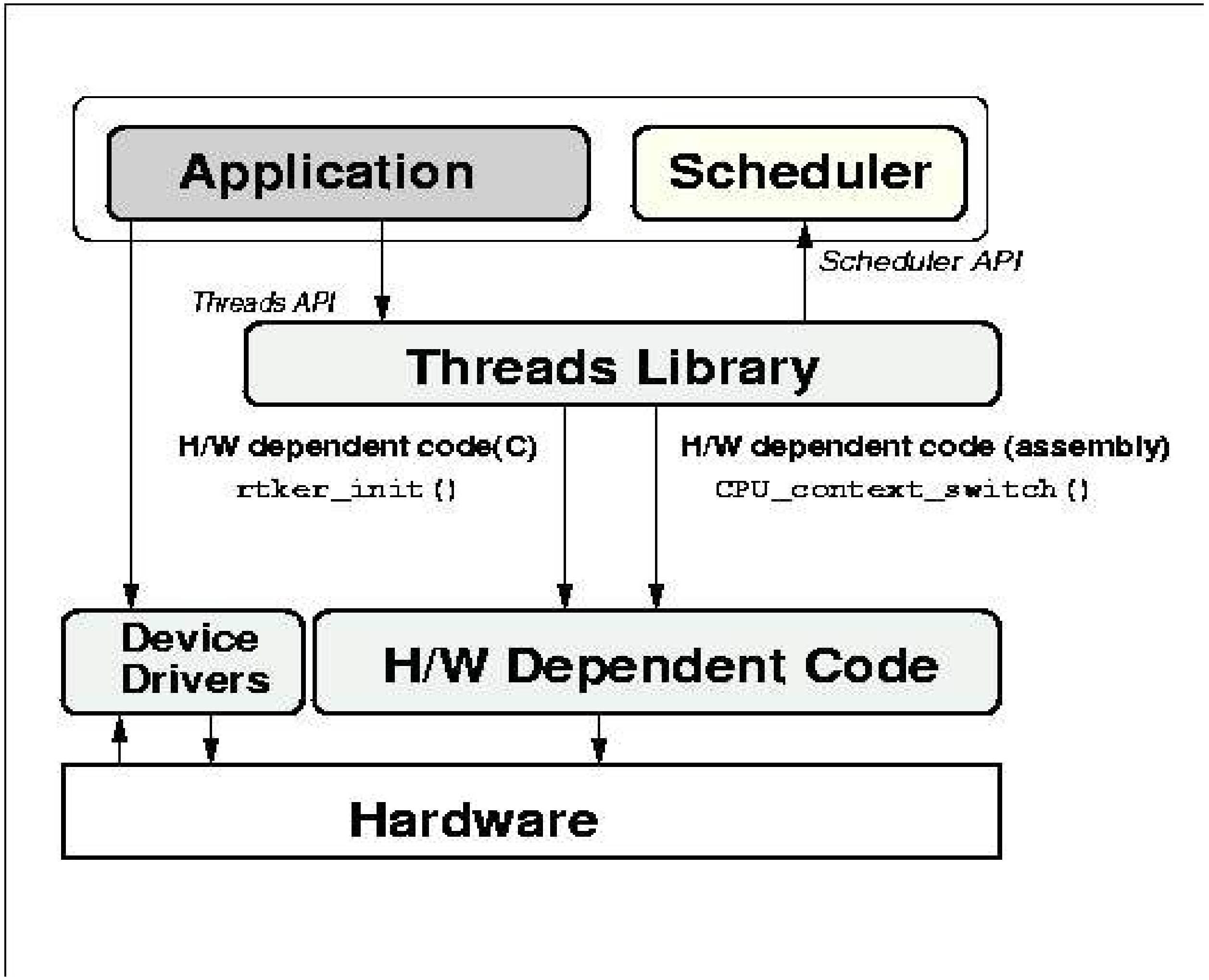
- Modular Design
  - Isolation of Architecture/CPU dependent and independent code – Easy to Port
- Pluggable Scheduler
- Two level Interrupt Handling
- Small footprint
- Oskit’s Device Driver Framework

Pluggable Scheduler

- Scheduler - part of the Application
- Kernel interacts with the scheduler through an API
- Application developer needs to implement the scheduler API
  - Can optimize on Data Structures & Algorithms for implementing the scheduler

- Can optimize on scheduling overheads.
- The kernel interacts with the scheduler through a set of functions.
- Whenever a new thread is created the kernel calls a scheduler function passing as argument the newly created thread.
- Subsequent scheduling of the thread & of all other threads in the system is for the scheduler to decide.
- At each context switch time (on a timer interrupt) the kernel calls a scheduler function to find out which thread to schedule next.

Block Diagram



Interrupt Handling in a real time operating system and a general purpose operating system .

- Minimum possible interrupt latency.
- Bounded maximum interrupt latency.

The delay in execution of the **Interrupt Service Routine(ISR)** from the time the device raises the interrupt is called interrupt latency.

Interrupt latency arises mainly because of the three factors -

- Kernel's disabling the interrupts to protect the kernel data structures from the interrupt service routines
- Kernel's processing time to call the ISR
- Device driver's disabling of interrupts because device specification required it.

Two Level Interrupt Handling

- Two level Interrupt Handling
  - Top Half Interrupt Handler
    - Called Immediately – Kernel <u>never</u> disables interrupts
    - Cannot invoke thread library functions - Race Conditions
  - Bottom Half Interrupt Handler
    - Invoked when kernel not in Critical Section
    - Can invoke thread library functions
- Very Low Response time (as compared to Linux)

Rtker supports interrupt handling mechanism wherein it remains responsive to the device while deferring accessing to the shared data structures.

Immediate ISR

- kernel vectors to it in case of an interrupt
- may run with interrupts disabled
- should not call any kernel library function
- Typically it acknowledges the device about the interrupt.

Bottom Half ISR

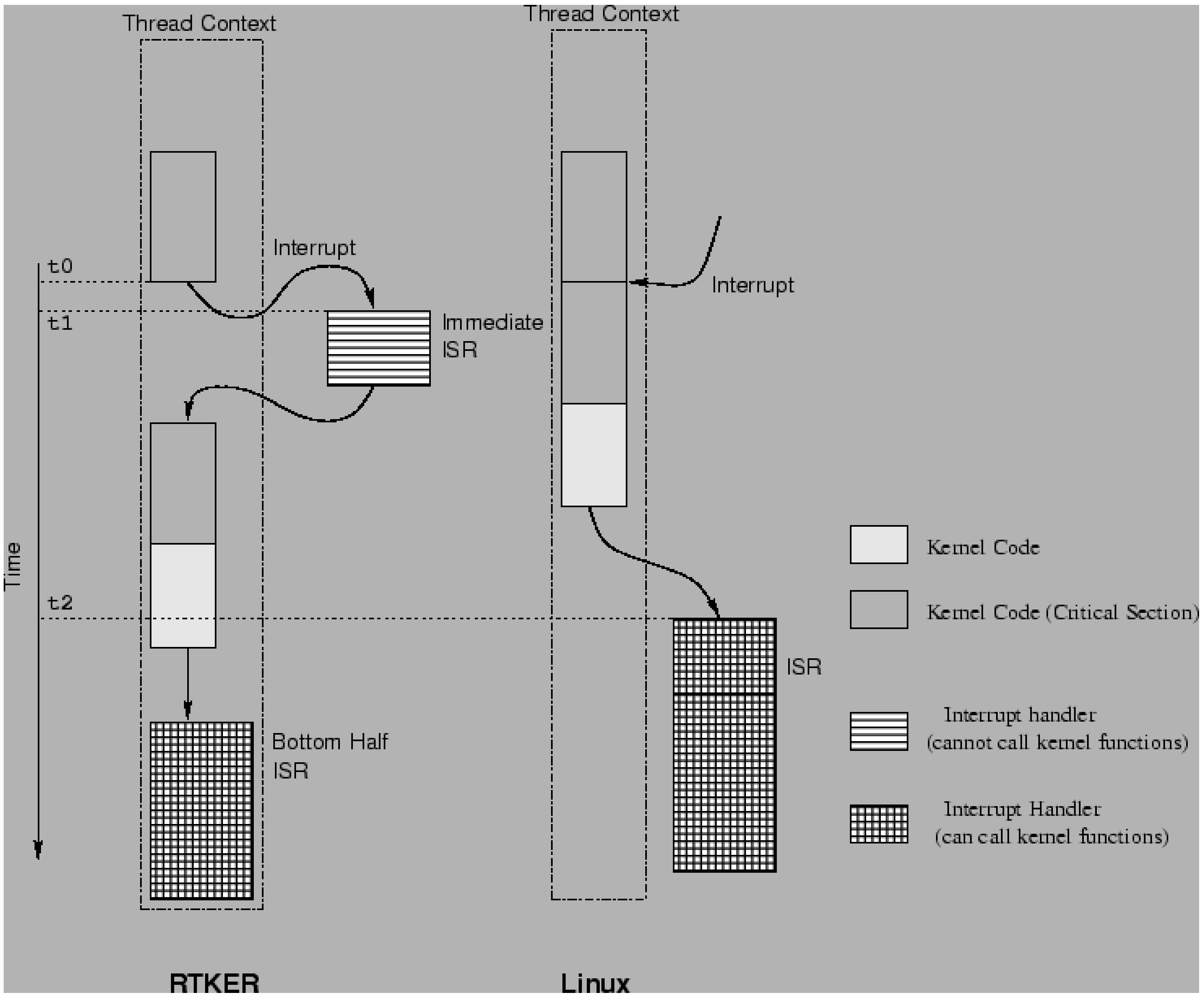
- may be activated from the Immediate ISR
- scheduled in a thread context by a special thread called the bottom half thread
- it runs in the context of a different thread and is free to make any kernel library call.
- Typically it does data transfers to/from the kernel.

Kernel Level Synchronization

The following dummy code illustrates how synchronization while accessing a shared kernel data structure is done without disabling interrupts.

```
oskit_u32_t dummy(void)
 {
   do_Context_Switch=0

        /*
    ................................
     Access any shared kernel data structure
    ................................
   */
   do_Context_Switch=1;
 }
```


Thread Context
Thread Context
Interrupt
t0
t1
Immediate ISR
Interrupt
Time
t2
ISR
Bottom Half ISR
Kernel Code
Kernel Code (Critical Section)
Interrupt handler (cannot call kernel functions)
Interrupt Handler (can call kernel functions)
RTKER
Linux

rtker's Device Driver Framework

- Motivation --> easy portability of existent device drivers.
- OSkit's Device Driver Framework has been used for *rtker*. Rtker has provided implementation of all the functions of OSkit's Device Driver Framework. These functions in turn call the *rtker's* functions for thread synchronization.
- Using the glue codes provided by OSkit, any Linux Device Driver can be incorporated into this RTOS without any change .

Thread Library

```
struct thread_info {
        void *(*run) (void *);
        void *arg;
        unsigned int type;          //PERIODIC, APERIODIC, RECOVERY,
                                    //IDLE_THREAD, BOTTOM_HALF thread.
        unsigned int stack_size;
        int readyTime; int execTime;
        int deadline;
        struct thread_info *recov_info; }
```

Thread Creation:	struct tcb *thread_create(struct thread_info *tinfo)
Thread Suspend:	unsigned int thread_suspend(struct tcb *t)
Thread Resume:	unsigned int thread_resume(struct tcb *t)
Thread Reset:	unsigned int thread_reset(struct tcb *t)
Thread Self:	struct tcb * thread_self(void)

Semaphore Library & Scheduling API

Typical Real Time Operating Systems support various priority handling policies, like priority ceiling and priority inheritance.

In rtker, the scheduling of threads (and hence priorities) is not handled by the kernel but by the user defined scheduling algorithm.

Functions like get priority and set priority in the Scheduler API can be invoked by the semaphore library for implementing the desired priority control policy.

Thread Control Block

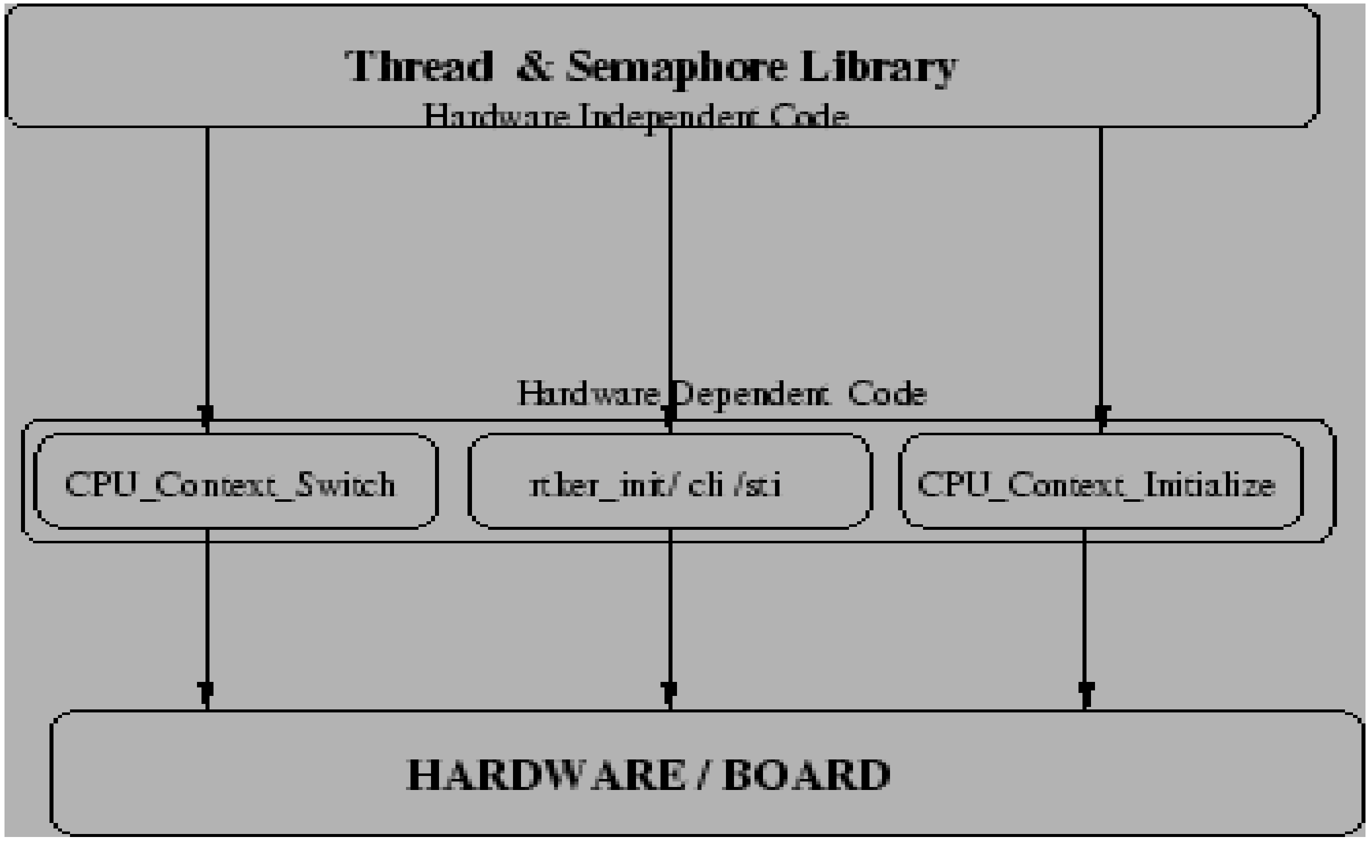
```
struct tcb {
  u32_t status; // THREAD_ON_CPU, THREAD_READY,
    THREAD_SUSPENDED THREAD_BLOCKED, THREAD_EXITED or
    THREAD_MISSED_DEADLINE.
  struct reg_context thread_context;
  void *stack;
  struct thread_info thread_params;
  u32_t executedTime;
  struct tcb *recoveryTask;
  u32_t sched_field;
  u32_t magic_key;
};
```

Scheduler

Set of functions which the application developer is expected to code is a part of the sched_info structure . A pointer to the structure is passed as argument to rtker_init() , during rtker initialization.

```
struct sched_info{
  spinlock_t sched_lock;
  oskit_u32_t (*init)(struct tcb *main_tcb);
  oskit_u32_t (*new_thread)(struct tcb *);
  struct tcb *(*heir_thread)(struct tcb *);
  oskit_u32_t (*set_mode)(struct tcb *,oskit_u32_t);
  oskit_u32_t (*reset_thread)(struct tcb *);
  oskit_u32_t (*delete_thread)(struct tcb *);
  oskit_u32_t (*set_priority)(struct tcb ,oskit_u32_t);
  oskit_u32_t (*get_priority)(struct tcb *);
  void (*tick)(void);}
```

Modularity



The thread and the semaphore libraries are totally hardware independent. They interact with the hardware dependent code through three functions and some macros.

```
void CPU_Context_Initialize(struct reg_context *the_context,void
*stack_base,
                        unsigned int stack_size,
                        void *(*entry_point)(void *,void *),
                        void *(*thread_fun)(void *),
                        void *thread_arg);
```

```
void CPU_Context_Switch(void *curr_thread, void *heir_thread)
```

```
int rtker_init(struct sched_info *sched);
```

This is the function for initialization of the Real Time Kernel. The argument passed is the pointer to the scheduler structure. This function is also responsible for setting up the interrupt tables, setting up the timer, registering the timer interrupt service routine etc

Thread Init Function

```
void *thread_init_function(void *(*start_add)(void *),void
   *param)
 {
   spin_unlock(&(scheduler->sched_lock));
   do_Context_Switch=1;
   start_add(param);
   do_Context_Switch=0;
   spin_lock(&(scheduler->sched_lock));
 ....... Re initialize the context of the thread ........
   scheduler->set_mode
   (curr_thread,THREAD_EXITED); ......... Context Switch
   to a new thread .........
}
```

Schedule Function

```
void schedule(void) {
 ..................... Check the Bottom Half .....................
    do_Context_Switch = 0;
    spin_lock(&(scheduler->sched_lock)); -------------(1)
    heir_thread=scheduler->heir_thread(curr_thread);
     if (!heir_thread) { heir_thread=idle_thread; }
     if (heir_thread!=curr_thread) {
          struct tcb *out_thread=curr_thread;
          curr_thread=heir_thread;
          CPU_Context_Switch(&(out_thread->thread_context), &
    (heir_thread->thread_context));
                                                  }
    spin_unlock(&(scheduler->sched_lock)); ----------(2)
    do_Context_Switch=1;
}
```